CYCLIC REDUNDANCY CHECKS IN USB

Introduction

The USB specification calls for the use of Cyclic Redundancy Checksums (CRC) to protect all non-PID fields in token and data packets from errors during transmission. This paper describes the mathematical basis behind CRC in an intuitive fashion and then explains the specific implementation called for in USB. Perl programs to generate these CRCs are provided and several examples are used to clear up possible ambiguous areas.

Two-minute mathematical background

The well-known concept of integer division forms the basis for the use of CRCs. When a dividend is divided by a divisor, a quotient and a remainder (which may be 0) result. If the remainder is subtracted from the dividend and this is divided by the divisor, the remainder is always zero. e.g. when 629 is divided by 25, the remainder is 4. If the dividend (629) and the remainder (4) are transmitted from a source to a target, the integrity of the transmission can be verified at the target by recomputing the remainder and verifying that the remainder matches the transmitted remainder. Alternatively, the target could divide the difference between the transmitted dividend and remainder and expect to see a zero remainder if there were no errors.

The concept of integer division can also be applied to division of polynomials. (An intuitive way to understand this is by considering that the digits which make up an integer can be considered the coefficients of a polynomial in base 10 e.g. $629 = 6*10^2 + 2*10^1 + 9*10^0$.). A binary bitstream (which is a pattern of 1s and 0s) can be considered to represent the coefficients of a (dividend) polynomial. When this polynomial is divided by a generator (divisor) polynomial (which is another binary bitstream) a remainder polynomial (CRC) will result. The arithmetic is especially simplified if 1 and 0 are considered to be elements of a finite field (the Galois field of order 2 or GF(2)) . The arithmetic is sometimes referred to as modulo 2 arithmetic. For the purposes of CRC computation, it is sufficient to understand that addition and subtraction in this field reduce to simple XOR operations. This is the basis of the technique used for generating and checking CRC for USB packets.

CRCs are useful because they are capable of detecting all single and double errors and many multiple errors with a small number of bits. Communications protocols often use two CRCs in a packet - one to protect the header of the packet and another to protect the data portion of the packet. In USB the header of the packet is the PID field. Since this field is only 4 bits long it is protected by a 4 bit check field derived by simple bitwise inversion of the PID field. This provides adequate single-bit and burst error protection

without requiring CRC generation and checking logic. The 'data' portion of a USB packet which is longer is protected by a conventional CRC field. In token packets, the CRC protected region is only 11 bits - so a 5 bit CRC provides adequate protection and also aligns the packet to a byte boundary. Data packets may have upto 1023 bytes; so a longer (16 bit) CRC is used to protect the data.

CRC implementation for USB

The USB spec lists two generator polynomials - one for tokens and the other for data packets. The generator polynomial for tokens is $x^5 + x^2 + x^0$ while the generator polynomial for data packets is $x^{16} + x^{15} + x^2 + x^0$. Since the remainder is always of smaller degeree than the generator polynomial, the token CRC is a 5 bit pattern and the data CRC is a 16 bit pattern.

An intuitive way to generate the CRC for an input pattern would be to simply divide this pattern by the generator poylnomial. The division can use the same approach used in integer division. If the Most Significant digit/Bit (MSB) of the dividend is a 1, the divisor is subtracted from the dividend to generate a new dividend. In modulo 2 arithmetic, this subtraction is simply a bit-wise XOR. The same step is repeated for the next MSB in sequence through all the bits including the Least Significant Bit (LSB). The remainder at this point is the remainder from dividing the input pattern multiplied by x^d by the generator polynomial of degree d (d is 5 for token CRC and 16 for data CRC).

The implementation called for in the USB spec differs from the above intuitive approach in three ways which are mathematically insignificant.

1. The implementation starts off with the shift register loaded with all 1s. Without this, leading 0s in front of a packet would not be protected by the CRC generated. Mathematically this is equivalent to adding a scaled constant to the dividend. The scaling is a function of number of bits in the input pattern. At the receiving target, the shift register is likewise primed with 1s. This ensures that the same scaled constant is added to the dividend at the target (assuming no bits are lost in transit). So if no bits are corrupted in transit, the same remainder will be generated at both ends. In equation form the scaling creates the dividend $D(x) = x^{32}F(x) + x^kL(x)$ where F(x) is a degree (k-1) polynomial representing the k bits of the data stream and L(x) is a degree (d-1) polynomial with all coefficients equal to one and d is degree of the generator polynomial.

2. The implementation uses commutativity and associativity of the XOR operation to advantage. In the intuitive approach, the new dividend is derived by subtracting the divisor from the dividend. In the implementation, this subtraction is done bit by bit on the dividend by accumulating and shifting the remainder.

3. The remainder is bit-wise inverted before being appended as the checksum to the input pattern. Without this modification, trailing zeros at the end of a packet could not be detected as data transmission errors. Mathematically, this is equivalent to adding a known

constant to the remainder. Again this is mathematically insignificant to the operation of CRC. In equation form, CRC= L(x) + R(x) where R(x) is remainder obtained by dividing D(x) by the generator polynomial G(x).

Checking the CRC at the target is the same as generating the CRC on an input pattern which now consists of the original input pattern followed by the inverted remainder. Mathematically, this new polynomial should be perfectly divisible by the generator polynomial except for the residual due to the known constant discussed in item 3 above. (This can be intuitively understood by realizing that the appending of the remainder to the LSB of the dividend is equivalent to subtracting it from the old dividend). In equation form, the transmitted and received data is $M(x)=x^{32}F(x) + CRC$. When CRC is generated on this pattern M(x), the remainder R'(x) is $x^{32}L(x)/G(x)$ and can be derived from the above equations and some properties of modulo 2 arithmetic. R'(x) is a unique polynomial (i.e. coeffiecients are always the same) since L(x) and G(x) are unique. R'(x) is termed the residue or residual polynomial.

For the token generator polynomial, the residual is 01100 (or $x^4 + x^3$) ; for the data CRC polynomial the residual is 1000000000001101 (or $x^{15} + x^3 + x^2 + x^0$).

Programs for generating CRC

While the above sections contain enough information to implement the CRC logic, it is quite easy to get the bit order confused. The following two programs in Perl (crc5 and crc16) can be used for checking the implementation of the token crc and data crc respectively.

```
#! /usr/local/bin/perl
## crc5 nrzstream
## e.g.  crc5 1000111
## nrz stream is sent in left to right order
## generated crc should also be sent out in left to right order

sub xor5 {
   local(@x) = @_[0..4];
   local(@y) = @_[5..9];
   local(@results5) = ();
   for($j=0;$j<5;$j++)  {
      if (shift(@x) eq shift(@y)) { push(@results5, '0'); }
      else { push(@results5, '1'); }
                  }
   return(@results5[0..4]);
}

            {
   local($st_data) = $ARGV[0];
   local(@G) =    ('0','0','1','0','1');
   local(@data) = split (//,$st_data);
   local(@hold) = ('1','1','1','1','1');
   if (scalar(@data) > 0)     {
   loop5: while (scalar(@data) > 0)       {
```

```
$nextb=shift(@data);
if (($nextb ne "0") && ($nextb ne "1")) {next loop5} ## comment
character
if ($nextb eq shift(@hold)) {push(@hold, '0')}
else  { push(@hold, '0'); @hold = &xor5(@hold,@G); }
##   print (@hold); print "\n";
                              }
                         }
##   print (@hold); print "\n";
## invert shift reg contents to generate crc field
for ($i=0;$i<=$#hold;$i++) {if (@hold[$i] eq "1") {print("0")} else {
print("1")} }
print "\n";

           }


#! /usr/local/bin/perl
## usage:
## crc16 nrzstream
## nrz stream is sent in left to right order
## generated crc should also be sent out in left to right order

sub xor16 {
   local(@x) = @_[0..15];
   local(@y) = @_[16..31];
   local(@results16) = ();
   for($j=0;$j<16;$j++)       {
      if (shift(@x) eq shift(@y)) { push(@results16, '0'); }
      else { push(@results16, '1'); }
                         }
   return(@results16[0..15]);
}
           {
   local($st_data) = $ARGV[0];
   local(@G) =
('1','0','0','0','0','0','0','0','0','0','0','0','0','1','0','1');
   local(@hold) =
('1','1','1','1','1','1','1','1','1','1','1','1','1','1','1','1');
   local(@data) = split (//,$st_data);
   if (scalar(@data) > 0)     {
   loop16: while (scalar(@data) > 0)      {
$nextb=shift(@data);
if (($nextb ne "0") && ($nextb ne "1")) {next loop16} ## comment
character
if ($nextb eq shift(@hold)) {push(@hold, '0')}
else  { push(@hold, '0'); @hold = &xor16(@hold,@G); }
                              }
                         }
#   print (@hold); print "\n";
## invert shift reg to generate CRC field
for ($i=0;$i<=$#hold;$i++) {if (@hold[$i] eq "1") {print("0")} else {
print("1")} }
print "\n";
           }
```

Examples of CRC computation

To further solidify the understanding of the implementation, the following examples are presented.

Consider generating an SOF token with timestamp of 710(hex). CRC5 can be used to generate the CRC on the timestamp with the following results.

```
crc5 00001000111
10100
```

The timestamp and CRC and EOP (End Of Packet) can be appended to the Sync, PID and PID check fields to generate the NRZ (Non Return to Zero) packet as follows.

```
00000001101001010000100011110100XX1;
```

In this example the NRZ sync field is 00000001 and the EOP is indicated as XX1 where X indicates a single ended 0. The PID and PID check fields are 0101 and 1010 respectively.

Note that bit stuffing and conversion to NRZI (Non Return to Zero Invert) will be performed (described in sec. 7.1.5 and sec. 7.1.6 of the spec) before the packet is transmitted on the USB. Since these operations do not affect the bit order or the CRC generation/checking they are not described in this paper.

Examples of SETUP, OUT, IN and SOF tokens (with address, endpoint and timestamp values in hex), the crc5 generation and the resulting NRZ packet are presented below.

```
setup addr 15 endp e

crc5 10101000111
10111

00000001101101001010100011110111XX1;

out addr 3a endp a

crc5 01011100101
11100

00000001100001110101110010111100XX1;

in addr 70 endp 4

crc5 00001110010
01110

00000001100101100000111001001110XX1;

sof timestamp 001

crc5 10000000000
10111
```

```
00000001101001011000000000010111XX1;
```

Examples of data packets are presented below. DATA0 and DATA1 packets, each with 4 bytes of data (indicated in hex) , the crc16 generation and the resulting NRZ packets are listed respectively below.

```
data0 00 01 02 03

crc16 0000000010000000010000011000000
1111011101011110

00000001110000110000000010000000010000001100000011110111010111 10XX1;
```

```
data1 23 45 67 89

crc16 1100010010100010111001101001 0001
0111000000111000

000000011101001011000100101000101110011010010001011100000011100 0XX1;
```